TOPIC 4

DESIGNING WITH PARALLEL STREAMS

PARALLEL STREAMS

RUNNING PARALLEL STREAMS

We can run any stream in parallel with `parallelStream()` from a `Collection` or `parallel` from a `Stream`

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble()
```

Source: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.htm

RUNNING REDUCTION SERIALLY

All Streams are serial by default

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

Source: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.htm

RUNNING REDUCTION CONCURRENTLY

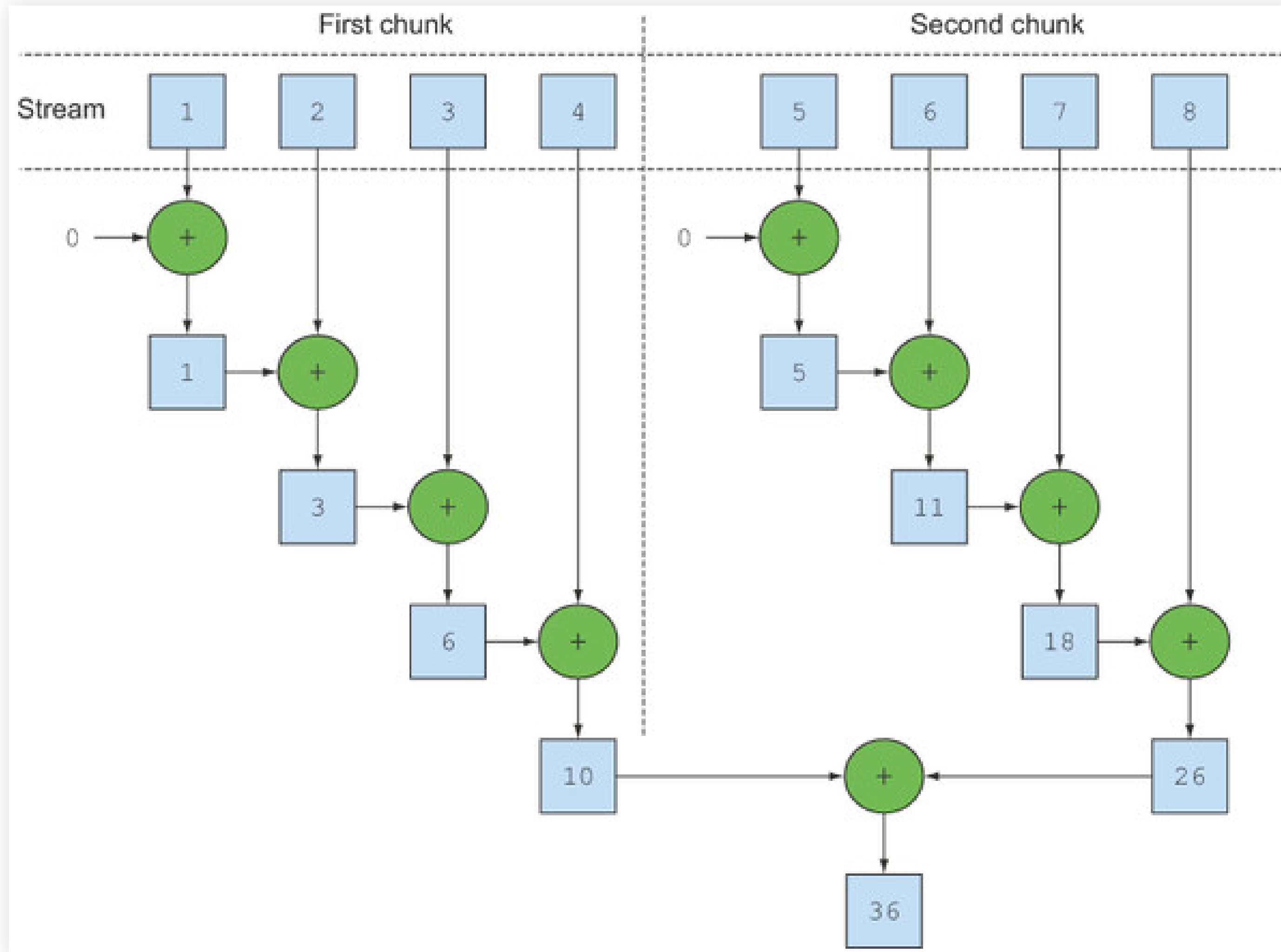
```
ConcurrentMap<Person.Sex, List<Person>> byGender =
    roster
        .parallelStream()
        .collect(
            Collectors.groupingByConcurrent(Person::getGender));
```

Source: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.htm

RUNNING REDUCTION CONCURRENTLY DIFFERENT EXAMPLE

```
public static long parallelSum(long n) {
  return Stream.iterate(1L, i -> i + 1)
               .limit(n)
               .parallel()
               .reduce(0L, Long::sum)
}
```

DIAGRAM OF PARALLELISM



Source: Java 8 In Action

NOTES ON CONCURRENT OPERATIONS

- The stream is parallel.
- The parameter of the collect operation, the collector, has the characteristic `Collector.Characteristics.CONCURRENT`. To determine the characteristics of a collector, invoke the Collector.characteristics method.
- Either the stream is unordered, or the collector has the characteristic `Collector.Characteristics.UNORDERED`. To ensure that the stream is unordered, invoke the `Stream.unordered` operation.

Source: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.htm

MAKING IT SEQUENTIAL AGAIN

Any Stream can be made `sequential`

```
stream.parallel()
       .filter(...)
       .sequential()
```

Source: Java 8 In Action

LAST CALL WINS GLOBALLY

- Last call to parallel or sequential wins and affects the pipeline globally
- This Pipeline will be executed in parallel because that's the last call in the pipeline

```
stream.parallel()
      .filter(...)
      .sequential()
      .map(...)
      .parallel()
      .reduce();
```

Source: Java 8 In Action

FORK JOIN POOLS

- Split a Parallelizable task into smaller tasks
- An implementation of the `ExecutorService` interface
- Distributes those subtasks to worker threads in a thread pool, called `ForkJoinPool`

`RecursiveTask` AND `RecursiveAction`

- `RecursiveTask<R>` is used to represent a task that returns a value
- `RecursiveAction` if it returns no result
- Tasks must implement one method:

```
protected abstract R compute();
```

RECURSIVELY CREATE SMALLER TASKS

```
if (task is small enough or no longer divisible) {
    compute task sequentially
} else {
    split task in two subtasks
    call this method recursively possibly further splitting each subtask
    wait for the completion of all subtasks
    combine the results of each subtask
}
```

Source: Java 8 In Action

DEMO: HOW IT SPLITS

Visit the Java 8 In Action Code at https://bit.ly/3s5uz9R

DEMO: COLLECTING IN PARALLEL

Visit the Repository, and we can do this together

PARALLEL SAFE CODING

CODING PROBLEM

Let's say that you need to process the values of a list of transactions by accumulating them into a particular bank account. The class `Account` provides three simple methods to process a transaction, add a certain amount to the total balance, and return the total balance.

Source: https://www.oreilly.com/content/solving-a-parallel-streams-puzzler-in-java-8/

Account

```
class Account {
   private long total = 0;
   public void process(Transaction transaction) {
       add(transaction.getValue());
   }
   public void add(long amount) {
       total += amount;
   }

   public long getAvailableAmount(){
       return total;
   }
}
```

PROCESSING TRANSACTIONS THE CLASSIC WAY

Let's say you have a list of transactions objects available, you can simply iterate through the list and process each transaction one by one using your bank account:

```
Account myAccount = getBankAccountWithId(1337);
for(Transaction transaction: transactions) {
   account.process(transaction);
}
System.out.println(myAccount.getAvailableAmount());
```

INSPIRED BY THAT YOU WANT TO USE STREAMS

What's the problem here? You are modifying the state of the account using an inherently sequential approach (i.e., you are iteratively updating its state.)

```
transactions.stream()
           .forEach(myAccount::process);
System.out.println(myAccount.getAvailableAmount());
```

THE PROBLEM WITH PARALLELISM

```
List<Transaction> transactions
   = LongStream.rangeClosed(0, 1_000)
               .mapToObj(Transaction::new)
               .collect(toList());
```

PRINTING THE OUTPUT WITH THE TRANSACTIONS

```
transactions.parallelStream()
           .forEach(myAccount::process);
System.out.println(myAccount.getAvailableAmount());
```

DIFFERENT RESULTS

You will find out that you will get different results with different execution

```
The total balance is 448181
The total balance is 421258
The total balance is 398291
```

This is very far off the correct result, which is 500500! In fact, what is happening is that you have a data race on each access to the field total. Multiple threads are trying to read, modify, and update the shared state of the bank account. As a consequence, they are stepping on each other's toes which leads to unpredictable outputs.

POSSIBLE SOLUTION?

You may be tempted to simply refactor the add method to be synchronized. But this is a bad solution because it adds further thread contention. In other words, your threads are waiting on the result of another before they can proceed. Although using `AtomicLong` doesn't require a global lock, the same principle remains: you want to let threads work independently without waiting on one another.

STREAM API EXPECTATIONS

The Streams API is designed to work correctly under certain guidelines. In practice, to benefit from parallelism, each operation is not allowed to change the state of shared objects (such operations are called side-effect-free). Provided you follow this guideline, the internal implementation of parallel streams cleverly splits the data, assigns different parts to independent threads, and merges the final result.

IDIOMATIC STREAM API

```
long sum = transactions.parallelStream()
                       .mapToLong(Transaction::getValue)
                       .sum();
myAccount.add(sum);
System.out.println(myAccount.getAvailableAmount());
```

CONCLUSION

While it may look appealing to use parallel streams because it is very simple to do so (after all, it's only a `parallel()` or `parallelStream()` call), code that works sequentially can often not work as expected in parallel. In addition, using parallel streams doesn't guarantee that the code will run any faster (it may actually run slower!) There are many caveats to consider including computation cost per element, size of the data, and characteristics about the data source.

INTERFERENCE

- Lambda expressions in stream operations should not *interfere*.
- Interference occurs when the source of a stream is modified while a pipeline processes the stream.

DEMO: INTEFERENCE

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.concurrent.stream` package and open *StreamInterference.java*.

PARALLEL ORDERING

PARALLEL ORDERING RULE

- If our `Stream` is ordered, it doesn't matter whether our data is being processed sequentially or in parallel; the implementation will maintain the encounter order of the `Stream`.
- It may not maintain that order when processing as a terminal operation

REMOVING ORDER

- We can remove ordering at any time with `unordered`
- The reason is two-fold:
    - First, since sequential streams process the data one element at a time `unordered()` has little effect on its own.
    - When we called `parallel()`, however, we affected the output.

INTERMEDIATE OPERATIONS

Some of the operations will affect ordering:

- `sorted(..)`
- `empty`

NOTE ON ORDERING

- Stream operations use internal iteration when processing elements of a stream.
- When you execute a stream in parallel, the Java compiler and runtime determine the order in which to process the stream's elements to maximize the benefits of parallel computing unless otherwise specified by the stream operation.

TERMINAL OPERATIONS

- `forEach` will possibly not maintain order after parallelization
- `forEachOrdered` guarantees to maintain the order of the `Stream`.

DEMO: ORDERING

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.concurrent.streams` package and open *StreamOrdering.java*.

PARALLEL PERFORMANCE CONSIDERATIONS

CAREFUL WITH BOXED VALUES

- Watch out for boxing
- Automatic boxing and unboxing operations can dramatically hurt performance
- Java 8 includes primitive streams (`IntStream`, `LongStream`, and `DoubleStream`) to avoid such operations

Source: Java 8 In Action

OPERATIONS THAT PERFORM POORLY

- Operations naturally perform worse on a parallel stream than on a sequential stream
- `limit` and `findFirst` that rely on the order of the elements are expensive in a parallel stream
- `findAny` will perform better than `findFirst` because it isn't constrained to operate in the encounter order
- You can always turn an ordered stream into an unordered stream by invoking the method unordered on it
- e.g. if you need N elements of your stream and you're not necessarily interested in the first N ones,
    - calling `limit` on an unordered parallel stream may execute more efficiently than on a stream with an encounter order.

Source: Java 8 In Action

TAKE INTO ACCOUNT HOW THE UNDERLYING STRUCTURE DECOMPOSES

- `ArrayList` can be split much more efficiently than a `LinkedList`
  - Dividing an `ArrayList` doesn't require a traverse
- Primitive Streams can decompose very quickly
- Decomposition works well with object that are a `Spliterator`

Source: Java 8 In Action

COLLECTIONS WITH GOOD DECOMPOSITION

Source	Decomposibility
`ArrayList`	Excellent
`LinkedList`	Poor
`IntStream.range`	Excellent
`Stream.iterate`	Poor
`HashSet`	Good
`TreeSet`	Good

CHEAP OR EXPENSIVE TERMINAL OPERATIONS

- Consider if a terminal operation has a cheap or expensive merge step
- If it is expensive, measure, the cost of parallelism might be negated.

Source: Java 8 In Action